EEE3096S Practical 2 Report

Chris Whittaker	WHTCHR013
Emily Slettevold	SLTEMI002

Introduction

The practical aims to demonstrate how Assembly code can be used to interface with an STM board and perform basic operations, namely making use of LEDs, pushbuttons and time delays. This low-level code has a default mode of operation where the LEDs increment by one every 0.7 seconds. Holding down various pushbuttons should trigger various responses according to the programming. These include manipulating the time delay, changing the LEDs to increment by either 1 or 2 at a time, pausing the pattern altogether, or displaying a specific pattern (0xAA). It is imperative to have a thorough understanding of low-level language as well as register manipulation to successfully complete the practical.

Methodology

- Import a new project into the STM IDE app using the 'practical 2' folder from the cloned gitHub repository. Then open the 'assembly.s' file to begin coding.
- Write code for a branch called 'delay_loop:' that implements a busy wait loop using a loop counter value that can be set using a register (r7 is used). This value corresponds to a specific time taken to complete the delay loop (time taken to count done to 0 from that number). Set the literals 'LONG_DELAY_CNT' and 'SHORT_DELAY_CNT' to the values corresponding to a 0.7 second and 0.3 second delay respectively. These will be used to set the register (R7) later. To calculate the values of the counter, take the frequency and multiply by 0.7 and 0.3 to get the number of cycles corresponding to those times, then divide by 2 to get the number of loops needed. This is because the implemented delay loop code takes 2 cycles to run a single loop so dividing by 2 gives the number of loops (groups of 2 cycles) and thus the loop counter value. $\frac{(0.7 \cdot 8000000)}{2} = 2800000$ for 'LONG_DELAY_CNT' and $\frac{(0.3 \cdot 8000000)}{2} = 1200000$ for 'SHORT_DELAY_CNT'.
- Write code for a branch to check the buttons' states ('check_buttons:'). Do this by saving the value of the IDR of GPIOA into a register while also saving the value corresponding to a certain button being pressed into another register and then comparing these registers. Depending on if a button was pressed or not, set the value of the register being used to store flags (r3 is used) as well as the register being used to store the delay loop counter (r7). Only change r7 to 'SHORT_DELAY_CNT' if PA1 is pressed. Also set the value of the register used to hold the LED pattern (r2 is used) when button PA2 is pressed. 0xAA should be written to r2 in this case because that is the pattern required to be displayed constantly during a PA2 press.
- Write code for a branch to set the pattern of the LEDs called 'write_leds:'. By default (when there is no flag set), the value of r2 should be incremented once and then written to the ODR of GPIOB. If the flag shows that PA0 was pressed it should increment r2 twice before writing to the ODR of GPIOB. If PA2 was pressed, the increments are skipped and r2 (with the value of 0xAA) is written to the ODR of GPIOB. The code checks if r2 has reached the value of 0xFF (all LEDs on) in which case it resets to a value of 0x00 (no LEDs on) and starts counting from zero again.

- Write the code for the ‘main:’ branch which controls the overall logic flow of the algorithm. First set r7 to be ‘LONG_DELAY_CNT’ above ‘main:’ so that this is used as the default delay time. Then under ‘main:’, set r3 to 0x00 to erase any flags from the previous loop. Branch to the ‘check_buttons:’ code. After branching back from ‘check_buttons:’, compare the value of r3 to the value of the flag set by a PA3 press. Use ‘BEQ main’ to go back to the top of the ‘main:’ branch and keep looping back to freeze the pattern as long as PA3 is being pressed. When PA3 has not been pressed, the code must move on and branch to ‘write_leds:’. After branching back to the main branch from ‘write_leds:’, the code should branch to ‘delay_loop:’ to cause a delay. Once the code has branched back to the ‘main:’ branch from ‘delay_loop:’ reset the value of r7 to ‘LONG_DELAY_CNT’ so this is used in the next loop by default. Then branch back to the top of ‘main:’ to cause the infinite loop.
- Flash the code onto the STM32F0 and check the functionality.

Results and Discussion

The practical yielded satisfactory results as all tasks can be successfully carried out on the STM32 board with the code developed. Upon reset, the LEDs increment by 1 every 0.7 seconds. Holding down various pushbuttons change the LED output and timing as required. SW0 being held down causes the LEDs to increase by 2 every 0.7 seconds. Holding down SW1 alters the delay between increments to be 0.3 seconds. The LED pattern can be set to 0xAA by holding down SW2, and to freeze the pattern, SW3 can be held down. After, the assembly code was flashed to the board, and the various pushbuttons were held down, it revealed that the code operated as expected. Additionally, SW0 and SW1 can be held down simultaneously to have the LEDs increment by 2 every 0.3 seconds as required.

Conclusion

Assembly code was successfully used to interface with the STM board and meet all the necessary requirements and functionality. The implemented code yielded satisfactory results. Busy wait delay loops were used to implement the delays. This method was chosen due to ease of implementation and simplicity. However, an alternative could be a more complex to implement timer interrupt that would be more accurate.

AI Clause

AI proved to be a useful tool during this assignment. The practical called for an in depth understanding of register manipulation in ARM Assembly. AI helped navigate the intricacies of register manipulation as well as the syntax of ARM Assembly, such as the difference between ‘CMP’ and ‘TST’ instructions. It highlighted the differences between the instructions and was able to recommend corrections as required. Hence, AI provided to be an efficient debugging tool with useful explanations.

Appendix:

Github: https://github.com/ChrisWhittakerUni/EEE3096S_Practicals_github

assembly.s code:

```
/*
* assembly.s
*
*/
@ DO NOT EDIT
.syntax unified
.text
.global ASM_Main
.thumb_func

@ DO NOT EDIT
vectors:
.word 0x20002000
.word ASM_Main + 1

@ DO NOT EDIT label ASM_Main
ASM_Main:

@ Some code is given below for you to start with
LDR R0, RCC_BASE          @ Enable clock for GPIOA and B by setting bit 17 and 18
in RCC_AHBENR
LDR R1, [R0, #0x14]
LDR R2, AHBENR_GPIOAB     @ AHBENR_GPIOAB is defined under LITERALS at the end of
the code
ORRS R1, R1, R2
STR R1, [R0, #0x14]

LDR R0, GPIOA_BASE        @ Enable pull-up resistors for pushbuttons
MOVS R1, #0b01010101
STR R1, [R0, #0x0C]
LDR R1, GPIOB_BASE  @ Set pins connected to LEDs to outputs
LDR R2, MODER_OUTPUT
STR R2, [R1, #0]
MOVS R2, #0  @ NOTE: R2 will be dedicated to holding the value on the LEDs

@ TODO: Add code, labels and logic for button checks and LED patterns
LDR R7, =LONG_DELAY_CNT @ store the address of the default delay cnt value to be
used in the delay_loop if no button has been pressed yet
main_loop:
MOVS R3, 0x0 @ reset R3 flag so the LEDs increment by only one at a time
```

```
BL check_buttons @ button check. If a specific button is being pressed the
necessary changes are made in 'check_buttons:'
MOVS R6, #0x02 @ put value that R3 will be if button PA0 was pressed into R6.
CMP R6, R3 @ check if R3 and R6 have the same values to freeze if PA0 was pressed
BEQ main_loop @ branch back to main_loop to avoid continuing on to write_leds.
This freezes the pattern because we do not increment in write_leds
BL write_leds @ do the next thing in the LED pattern
BL delay_loop @ delay between changes to LED pattern
LDR R7, =LONG_DELAY_CNT @ reset to default long delay
B main_loop
```

```
write_leds:
MOVS R6, #0x03 @ set R6 to check against R3 flag for PA2 press
CMP R6, R3 @ check if R3 is 0x03 - PA2 was pressed
BEQ write @ if PA2 was pressed, do not increment and just jump to the write
MOVS R6, #0xFF @ set R6 in order to check against R2
CMP R6, R2 @ check if R2 is 0b11111111
BNE normal_iteration @ if in the middle of sequence (R2 not equal to 0b11111111)
then branch to the normal iterating logic
MOVS R2, #0x0 @ if R2 was 0b11111111 set it to 0b00000000
B write @ skip over iterating part so 0b00000000 does not get set to 0b00000001

normal_iteration:
MOVS R6, #0x1 @ put the value that you need to compare to the '2 or 1' flag into
R6
CMP R3, R6 @ compare to see if '2 or 1' flag is set or not
BNE _inc_1 @ do an extra increment (2 at a time), if R3 has been set to 0x1
meaning PA0 was pressed
ADDS R2, R2, #0x1 @ increment the value in R2
_inc_1:
ADDS R2, R2, #0x1 @ increment the value in R2 a second time if necessary
write:
STR R2, [R1, #0x14] @ put appropriate leds on by writing R2 to GPIOB's ODR

done:
BX LR @ branch back to where it was before the branch to write_leds
```

```
check_buttons:
LDR R5, [R0, 0x10] @ read in IDR of GPIOA into R5
check_PA0:
MOVS R6, #0x01 @ store bitmask to compare IDR of GPIOA to to see if button 0 was
pressed
TST R5, R6 @ test IDR against the value that shows button on PA0 was pressed
```

```
BNE check_PA1 @ if not equal to the bitmask, PA0 was not pressed so skip over and check PA1
MOVS R3, 0x1 @ flag to say that you must increment by 2 LEDs every time not 1
check_PA1:
MOVS R6, #0x02 @ store bitmask to compare IDR of GPIOA to to see if button 1 was pressed
TST R5, R6 @ test IDR against the value that shows button on PA1 was pressed
BNE check_PA2 @ if not equal to the bitmask then PA1 was also not pressed so skip over and go to check PA2
LDR R7, =SHORT_DELAY_CNT
check_PA2:
MOVS R6, #0x04 @ store bitmask to compare IDR of GPIOA to to see if button 2 was pressed
TST R5, R6 @ test IDR against the value that shows button on PA2 was pressed
BNE check_PA3 @ if not equal to the bitmask then PA2 was also not pressed so skip over and go to check PA3
MOVS R2, #0xAA
MOVS R3, #0x03
check_PA3:
MOVS R6, #0x08 @ store bitmask to compare IDR of GPIOA to to see if button 3 was pressed
TST R5, R6 @ test IDR against the value that shows button on PA3 was pressed
BNE done_checking @ if not equal to the bitmask then PA3 was also not pressed so skip over and go to the end of the check
MOVS R3, #0x2 @ R3 set to value to flag a freeze

done_checking:
BX LR @ return after button checks have been done and the appropriate updates have been made



delay_loop: @ start of delay function
LDR R4, [R7] @ The address of either SHORT_DELAY_CNT or LONG_DELAY_CNT is stored in R4. Take this address and use it to load the actual value of the delay cnt (at this address) into R4
delay_loop_inner: @ start of loop
SUBS R4, R4, #1 @ subtract 1 from the value in R4 (decrement the counter)
BNE delay_loop_inner @ if R4 is still not at 0 redo the decrement else if it has reached zero move on
BX LR @ counter has reached zero so return back from the delay function



@ LITERALS; DO NOT EDIT
.align
RCC_BASE:                .word 0x40021000
AHBENR_GPIOAB:           .word 0b1100000000000000000
GPIOA_BASE:         .word 0x48000000
GPIOB_BASE:         .word 0x48000400
MODER_OUTPUT:            .word 0x5555
```

```
@ TODO: Add your own values for these delays
LONG_DELAY_CNT:     .word 0x2AAE80 @ value that counter of delay loop must count
till to have a delay of 0.7 seconds
SHORT_DELAY_CNT:    .word 0x124F80 @ value that counter of delay loop must count
till to have a delay of 0.3 seconds
```